Location-Aware Cache Management for Many-Core Processors with Deep Cache Hierarchy

Jongsoo Park[1], Richard M. Yoo[1], Daya S. Khudia[2], Christopher J. Hughes[1], and Daehyun Kim[1]

[1]Parallel Computing Lab, Intel Corporation, [2]University of Michigan - Ann Arbor

ABSTRACT

As cache hierarchies become deeper and the number of cores on a chip increases, managing caches becomes more important for performance and energy. However, current hardware cache management policies do not always adapt optimally to the applications behavior: e.g., caches may be polluted by data structures whose locality cannot be captured by the caches, and producer-consumer communication incurs multiple round trips of coherence messages per cache line transferred. We propose load and store instructions that carry hints regarding into which cache(s) the accessed data should be placed. Our instructions allow software to convey locality information to the hardware, while incurring minimal hardware cost and not affecting correctness. Our instructions provide a 1.07× speedup and a 1.24× energy efficiency boost, on average, according to simulations on a 64-core system with private L1 and L2 caches. With a large shared L3 cache added, the benefits increase, providing 1.33× energy reduction on average.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*

General Terms

Design, Experimentation, Performance

Keywords

Energy-Efficient Memory Hierarchy, Producer-Consumer Communication, Reuse Distance, Streaming Memory Accesses

1. INTRODUCTION

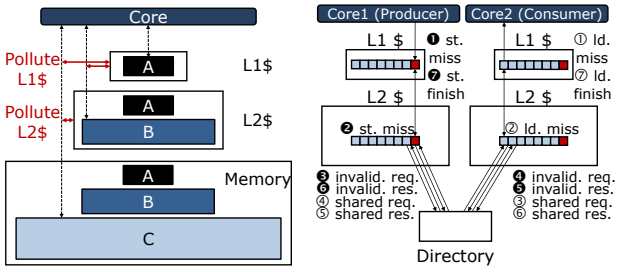
As cache hierarchies become deeper and the number of cores on a die increases, the penalty for suboptimal cache management (i.e., replacement policy, coherence protocol, etc.) increases. Ideally, hardware would automatically make the best use of the on-die memory capacity to maximize performance and minimize energy consumption; hardware should keep a copy of each piece of data in the right set of caches, and in the right coherence state, at every point of execution. However, this requires either predicting or otherwise understanding what data fits in which cache and will be used by which core, and when—as cache hierarchies become more complex, achieving efficient behavior is more difficult.

Many current hardware cache managers are particularly inefficient when (1) frequently accessed data structures stream through the cache hierarchy, and (2) producer-consumer communication takes place.

For (1), "streaming" access patterns typically occur when we sequentially sweep an array. We also observe streaming behavior when a specific cache level cannot capture the temporal locality of a data structure[1]. For example, in Fig. 1(a), three data structures `A`, `B`, and `C` fit in L1, L2, and main memory, respectively; `B` is streaming with respect to L1, and `C` with respect to L1 and L2.

Streaming access patterns can trigger significant cache hierarchy inefficiency. For our example, an optimal cache manager would separately place `A`, `B`, and `C` at L1, L2, and main memory, respectively. However, a cache manager with a conventional LRU replacement policy, or something similar, would try to place everything in L1, and fail to capture much temporal locality. Allocating cache lines that are not

[1]Sometimes streaming carries the narrow meaning of accessing data with no reuse. This corresponds to streaming with respect to the last level cache (LLC) in our broader definition of streaming.



(a) Streaming Pattern (b) Producer-Consumer Pattern

Figure 1: Hardware Cache Management Inefficiencies. On the right, the numbers denote the necessary coherence steps.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17-21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503224

reused unnecessarily consumes energy, and may also pollute the cache by evicting useful data.

For (2), as illustrated in Fig. 1(b), producer-consumer communication involves a thread writing a chunk of data, which is then read by other threads. With an invalidation-based coherence protocol, the hardware will behave as follows: for each cache line involved (e.g., lines containing dark dots), the producer finds and invalidates any existing sharers, brings the line into its cache(s), and performs the write. Then, the consumer must find the producer and grab a copy of the data from the producer's cache.

This process, or cache line "ping-ponging," is inefficient in multiple ways. First, since the producer will not read the data again after its write, there is no benefit to placing the line in any of the producer's caches (unless it shares a cache with the consumer). Second, if the producer-consumer pattern reuses a given cache line, having the producer invalidate the consumers before its write is unnecessary—in a correct program, a consumer will not read the line again until the producer has updated it.

Therefore, we propose *cache-location-aware memory instructions*. These instructions *allow software to provide application level knowledge to the hardware*. Since they are implemented via small *extensions to existing cache managers*, the hardware costs are minimal.

Specifically, we propose two new instructions: *unallocating load* and *pushing store* (in short, ULPS). An *unallocating load* indicates that a read is to a piece of data that should not be allocated in caches smaller than the working set size (more precisely, reuse distance [1, 12, 48, 50] of the data, whose definition will be described in Section 2). This is similar to streaming loads (e.g., `MOVNTDQA` in Intel 64 [22]), but provides finer-granularity control by allowing data to be cached at certain levels in the hierarchy. A *pushing store* is a write that pushes the data to a specific cache in the hierarchy—it provides control over not only *which level* of cache the data should be in (*local* pushing store), but also *which core* (e.g., a consumer thread's core) the data should go to (*remote* pushing store).

Unallocating loads and local pushing stores control the level at which data structures are allocated. They are particularly useful when all or a portion of a working set does not fit in a certain level of cache. For example, for the case shown in Fig. 1(a), we can use normal load/stores for `A`, ULPS with reuse distance hints larger than L1 for `B`, and ULPS with reuse distance hints larger than L2 for `C`. Such an optimization improves performance and energy efficiency, since it (1) increases cache hit rates through reduced pollution, and (2) elides wasteful allocations into the caches.

Remote pushing stores are designed for efficient inter-core communication between producers and consumers. Suppose that, in Fig. 1(b), the producer (core 1) issues a pushing store to the consumer (core 2). A pushing store pushes the cache line directly to the consumer's cache instead of invalidating the consumer's cache. Later, when the consumer reads the shared data, the updated line will already be in the consumer's local cache. Using pushing stores to perform one-way transfers improves performance and energy efficiency, since it (1) reduces the number of coherence messages, and (2) avoids latencies involved in cache line ping-ponging and coherence misses for the consumer.

There have been many previous works to address inefficiences of streaming and producer-consumer communication. However, our proposal is a unique solution that (1) naturally *extends the conventional cache architecture*, and thus requires minimal hardware cost, and (2) provides *a unified interface* to software to cover both cache-level management and producer-consumer communication seamlessly. Specifically, our proposed instructions have the following distinct advantages over other schemes.

1. Our instructions maintain the fundamental behavior of caches and simply extend software control over which cache(s) should hold the accessed data. In contrast to more disruptive proposals such as software-managed buffers [9, 18], our instructions merely "suggest" a preferred behavior for the hardware and preserve program correctness.

2. Our instructions provide more control over data placement. Streaming load/stores and data forwarding work [26] resemble some aspects of our instructions, but they provide limited control. For example, streaming load/stores provide a hint to the hardware that the data should not be placed in *any* cache; they cannot optimize streaming access data that fit some caches, and they do not help with inter-thread communication.

This paper not only shows that our unified approach to addressing both of these sources of inefficiency works well, it also demonstrates that our scheme is particularly efficacious on processors with *deep cache hierarchies*. Deep cache hierarchies, as exist on many-core processors, impose unique challenges/aspects that have not been previously addressed. In particular:

- **Energy is an important factor.** In modern many-core processors, energy and/or power is the most constrained resource. To the best of our knowledge, no previous work on cache management efficiency provides rigorous energy analysis. We quantify *both* the performance and energy efficiency improvements of our instructions (§5), and observe that the energy benefits of ULPS are greater than its performance benefits. On a 64-core simulated system, they provide on average 1.07× performance improvement, while simultaneously reducing the memory system energy by 1.24× on average.

- **Finer-grained control is necessary.** As noted, previous work, particularly in the cache level management area, provides only *coarse-grained* control. This inflexibility is increasingly problematic as the hierarchy deepens. On a two-level cache hierarchy, we see level-awareness brings 1.03× and 1.15× performance and energy improvements on average, respectively, over coarse-grained streaming load/stores. In addition, with a deeper hierarchy including a large, L3 embedded DRAM cache, streaming load/stores can not take advantage of the L3, even in cases where most or all of the working set fits. There, the ULPS advantage increases significantly, 1.11× and 1.33× performance/energy improvements on average.

- **Programmability/portability must be addressed.** Ease of use of a cache management scheme in a real program and not having to re-tune an application on a different cache hierarchy are important issues, especially for complex cache hierarchies. We provide a detailed discussion on how a compiler could generate our instructions (§4). Also, our instructions take reuse distance as a locality hint, and the actual cache levels to which the lines

should be brought in are determined by the hardware at run-time, providing performance portability.

2. INSTRUCTION INTERFACE AND USAGE PATTERNS

We propose two new instructions: *unallocating load* and *pushing store*. This section defines the semantics of these instructions and demonstrates how they can be used.

2.1 Instruction Specification

Our proposed instructions extend the existing semantics of load and store instructions, mostly with respect to desired cache locations. With simple hints, which is also optional, programmers can control data location within deep cache hierachies among multiple cores.

- `unallocating_load reg, mem, rd_reg`
  - `reg`: target register
  - `mem`: source memory address
  - `rd_reg`: register that holds reuse distance hint

- `pushing_store mem, reg, rd_reg, core_reg, time`
  - `core_reg`: register that holds target core hint (`self`, core ID, or `all_sharers`)
  - `time`: eager or lazy

Unallocating loads behave like normal loads, except they carry a hint to not allocate the data in caches smaller than the specified reuse distance [1, 12, 48, 50]. The reuse distance is the total size of distinctive data elements accessed between consecutive accesses to the memory address [12]. At run-time, the hardware compares the reuse distance hint with the capacity of each cache level to determine whether a cache line should be allocated in that level. Since the reuse distance is simply a hint, the software does not need to provide precise values to improve performance and energy. Tools such as auto-tuners can also help tune the values. Section 4.1 discusses how to use reuse distance for caches shared among multiple threads.

Similarly, pushing stores are the same as normal stores, except they hint to not allocate in caches smaller than the specified reuse distance. Additionally, pushing stores take a core ID as a hint so that lines can be "pushed" to the target core's cache. Pushing stores with `self` as a core hint are *local pushing store*s, while others are *remote pushing store*s. In addition to a specific core ID or `self`, the core hint may be `all_sharers`, which indicates that the data should be pushed to all the cores who currently share the corresponding line. We provide this simple yet powerful sharing hint, since for dynamic sharing patterns identifying consumer cores at compile time can be challenging. The time hint specifies when the line should be pushed: immediately (i.e., eager), for latency-critical writes, or after buffering (i.e., lazy), if we expect further writes to the same line.

2.2 Usage of ULPS

We now discuss when and how to use ULPS instructions to improve cache behavior. In particular, this section overviews the set of workloads used in this study; the detailed per-application analysis can be found in Section 5.

Table 1 shows the workloads, grouped into microbenchmarks and applications. In the table, the "Access Patterns" column denotes application patterns that will be described in this section. Our workloads exhibit varying patterns. Explicit cache levels are specified in some columns to indicate run-time behavior on our simulated system (see Section 5); for portability, our instructions use reuse distance. For example, UL>L1 for `Overflow` indicates that we use unallocating loads with reuse distance hint larger than L1 and smaller than L2.

2.2.1 Cache-Level Management

Our first set of patterns is related to managing cache levels, and in particular, in handling streaming accesses. An access pattern may be streaming with respect to one cache, but not to a larger one capable of holding the working set; thus, when explaining if data is streaming (and how), it must be with respect to a specific cache level. Programmers often have application-level knowledge of how large a cache should be to capture temporal locality of each data structure; ULPS can take advantage of this knowledge.

Three streaming situations where ULPS can help are:

1. Non-temporal: all data structures have no reuse.

2. Working Set Overflow (`WO`): one or more data structures see reuse, but the reuse distances are all too large for the cache.

3. Cache Pollution (`CP`): multiple data structures (or multiple pieces of the same data structure) see reuse, and at least one has a small enough reuse distance for the cache and at least one does not.

`Overflow` is a microbenchmark that demonstrates the `WO` pattern in its simplest form. Three arrays are repeatedly and sequentially accessed, and each is half as large as the L1-D. By using unallocating loads for one of the arrays, we can avoid wasteful cache allocations (i.e., placing a line in cache and evicting it before reusing it) and pollution.

For `Radix` sort [40], and `CG` and `BT` from NAS Parallel Benchmarks (NPB) [31], large input/output arrays stream from memory, while other smaller data structures exhibit locality patterns that can be easily captured throughout the cache hierarchy. This leads to a `CP` pattern at L2, where we can use ULPS to bypass L1 and L2 for accesses to the large arrays.

Our applications with more irregular accesses, `Backprojection` and `Hashjoin` [25], exhibit complex locality behavior. The L1 is too small to capture significant locality for any data structures; hence they exhibit a `WO` pattern at L1. The L2 can hold some data structures, but both applications have key arrays that cannot fit in L2, leading to a `CP` pattern at L2. Since these applications show different behavior for the different levels of cache, we expect ULPS's finer control over cache level (compared to streaming load/stores) to be beneficial.

`RTM` [2] is a stencil computation and exhibits multiple streaming and inter-core communication patterns. In `RTM`, some arrays are larger than the LLC, and are blocked into LLC-resident buffers. These large arrays pollute caches, for which we can use ULPS with a reuse distance hint larger than L2 (`CP` pattern at L2). `RTM` involves sweeping a 3-D filter over each buffer. Different data elements see different reuse distances, depending on how the sweep is done; some elements see reuse in the L1, but some do not (`CP` pattern at the L1). We use ULPS to reduce L1 pollution.

Table 1: Workload Descriptions*: WO = Working Set Overflow, CP = Cache Pollution, SN = Synchronization, and XF = Block Xfer. In "ULPS Application" column, UL>L1 indicates that unallocating loads are used with reuse distance hints larger than L1 and smaller than L2.

	Name	Description	Input	Access Pattern WO	CP	SN	XF	ULPS Application
uBench	Overflow	Sum elements of 3 arrays. Only two can fit in L1 at a time.	Three 16 KB arrays accessed 100 times	L1				UL>L1
	Barrier	Radix-4 combining tree barrier [51]. Threads communicate through flags.	100 iterations			✓		eager remote PS<L1
	Wr/Rd	Producer creates a linked list, and consumer chases the pointers.	L2-sized linked list				✓	eager remote PS>L1
App.	Radix	Radix sort with 8-bit radix [40]. Significant reuse on histograms.	32 M 32-bit integers		L2			UL>L2 for in/out
	CG	Conjugate gradient solver from NPB [31]. A sparse matrix is streamed over to access corresponding elements in a vector.	Class C		L2			UL>L2 for matrix
	BT	A block tri-diagonal solver from NPB. Significant reuse on coefficient data structures.	Class C		L2			UL>L2 for output
	Backproj-ection	Constructs a volume from 2-D projections. Locality exists for projections.	512^3 volume	L1	L2			UL>L1 for projections local PS>L2 for out
	Hashjoin	Join tables by streaming over inputs (outer-table) and looking up a hash table (inner-table) [25].	256 K-entry tables	L1	L2			UL>L1 for inner-table local PS>L2 for out
	RTM	Reverse time migration widely used in seismic imaging [2] with 3.5-D blocking optimization [32].	256^3 8-point stencil		L1, L2		✓	local ULPS>L1 and >L2 lazy remote PS>L1
	2D-FFT	Two phases, one with row-wise 1-D FFTs and the other with column-wise 1-D FFTs.	1K×1K SP-complex				✓	lazy remote PS>L1
	BigDFT	Computational chemistry workload [6]. 3D convolution with x/y/z-dimension repartitoning.	32^3 28-point stencil				✓	lazy remote PS>L1
	Stream Cluster	Clusters stream of input data chunk at a time. From PARSEC [5].	simmedium			✓		eager remote PS<L1

*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

2.2.2 Producer-Consumer Communication

Programmers are often aware of which hardware *threads* will produce and consume pieces of data and *when* they will be produced and consumed. By proactively pushing data to the consumer, we can (1) reduce the number of coherence messages to save energy, and (2) hide the communication latency.

Two common situations where producer-consumer communication is used and can have a large impact on performance and/or energy are: (1) synchronization operations (SN pattern), and (2) block data transfers between parallel sections (XF pattern).

Synchronization.

Synchronization operations commonly involve sending short messages between threads, such as in barriers and queue-based locks [41]. Typically, the message receiver spin-waits on a location in memory until the sender writes a particular value into that location. We can use a pushing store to more efficiently handle this communication.

Barrier is a microbenchmark comprised of a series of combining tree barriers [51], a popular barrier implementation. It has well-defined parent-children relations between threads, each of which acts as producer or consumer at each barrier phase. In invalidation-based cache coherence, cache lines are ping-ponged between between parents and their children (similar to Fig. 1(b)). It is more efficient to directly push the cache line to the consumer, with a pushing store.

StreamCluster is from the PARSEC [5] benchmark suite. The workload has frequent barriers to synchronize parallel sections, exhibiting an SN pattern.

Block Data Transfers.

A standard model of parallelizing a multi-step computation is as follows. First, partition the data set and assign each partition to a thread. Next, have each thread update its own partition using a single parallel section per computation step. Finally, between parallel sections, communicate any necessary updated data between threads.

This model is extremely common, for instance, in high performance computing applications, including most that use stencil algorithms such as RTM. In these stencil algorithms, the primary data structure represents a 2D or 3D space, and during each iteration, a thread reads values along its 2D or 3D boundaries that its neighbor threads computed in a previous iteration. When a thread reads the boundary values from its neighbors, they are likely to reside in their neighbors' caches.

In invalidation-based cache coherence, this communication requires two-way transfers (e.g., consumer → directory → producer → directory → consumer). If we use pushing stores, these become one-way transfers (e.g., producer → directory → consumer), saving significant latency and energy.

Another common usage of block data transfers is data repartitioning. In some cases, different steps of a computation require (or are faster with) a different partitioning of the data structure. For example, in a 2D-FFT, one phase performs computation with row-wise accesses while another works column-wise. Therefore, we first partition the data horizontally, then vertically. At the beginning of the second phase, most of the data accessed by each thread will reside in remote caches. At the end of the first phase, we can use pushing stores to push the data to consumers' caches. BigDFT [6] undergoes similar data repartitioning.

3. HARDWARE MODEL

We now describe a hardware implementation of unallocating loads and pushing stores.

3.1 Baseline System

Many implementations of our proposed instructions are possible. For the one we present here, we assume a tile-based architecture, where each tile has a core and private L1 and L2 caches. The tiles are connected through an interconnect, and cache coherence is maintained through a directory-based protocol (e.g., MOSI). The protocol supports clean line forwarding, so a miss in the L2 can be served either by a remote cache or memory. We also assume a consistency model similar to total store ordering (TSO) [42].

3.2 Implementation Overview

We implement ULPS by extending the load/store unit, the cache controller, and the directory controller. The new functionality does not modify the cache or directory behavior of any existing instructions, since our mechanism is only triggered by ULPS instructions.

Compared to normal loads, *unallocating loads* differ primarily in their cache line allocation policy: the cache allocates a line only when its capacity is equal to or smaller than the reuse distance hint. For example, suppose a core executes an unallocating load from address `A` with reuse distance hint 256 KB, and L1-D and L2 caches are 32 KB and 512 KB, respectively. If `A` is already cached in the L1, the L1 sends the data to the core. Otherwise, the L1 sends a request to the L2, but later sends the data returned from the L2 to the core without allocating a line.

Local pushing stores work similar to unallocating loads, except that they put the lines into a modified (dirty) state instead of shared (clean) state. To acquire the ownership of the cache line, we depend on the underlying coherence protocol. Assuming the same cache sizes, suppose a core executes a local pushing store to address `A` with reuse distance hint 256 KB. If `A` is already in modified state in the L1 cache, we perform the write in L1. If `A` is in the L1 but without write permission, the L1 controller first acquires ownership of the line, then performs the write. Otherwise, the L1 sends a request to the L2, and the L2 handles the request identically to a normal store, except that the write is done at the L2 and the corresponding line is not sent back to the L1.

For *remote pushing stores*, lines are pushed through the directory to maintain coherence. Suppose core 1 in Fig. 2 executes a pushing store with target hint of core 2 and reuse distance hint of 256 KB. We first send the pushing store request to the directory (step 1 in Fig. 2). In order to maintain coherence, the directory invalidates all sharers (step 2) and fetches the line with latest data (step 3), similar to when an ownership request is received. Then, the directory forwards the line and an update request to core 2 (step 4). Core 2 performs a local pushing store with reuse distance hint 256 KB on behalf of core 1 (this local pushing store is performed without additional directory lookups since the line is owned by core 2 due to invalidations performed in step 2).

When an `all_sharers` hint is used, the directory forwards update requests to all sharers in step 4, and each sharer performs a local pushing store with reuse distance hint 256 KB (again without additional directory lookups).

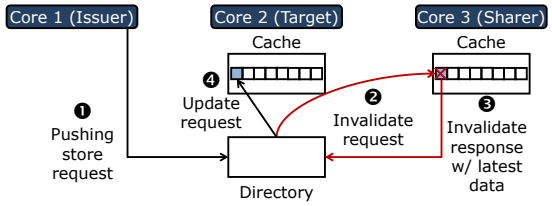


Figure 2: Remote Pushing Store. Core 1 issues a remote pushing store to core 2 when core 3 has the latest corresponding line.

3.3 Performance Optimizations

As discussed in Section 2.2, in real-world applications, producer-consumer communication is used for two different purposes: synchronization and data transfers. More importantly, these two scenarios dictate different latency constraints. When pushing stores are used for synchronization, such as in a barrier, it is critical to minimize the communication latency by immediately pushing data to remote caches. In contrast, for block data transfers, which typically take place between parallel sections, latency can often be hidden. Instead, it is important to accumulate data updates (i.e., combine multiple updates to the same cache line) to exploit spatial locality and save messages.

Therefore, we implement two versions of remote pushing stores: eager and lazy. *Eager remote pushing stores* immediately send messages to the directory, while *lazy remote pushing stores* accumulate data in streaming store buffers before sending messages.

A streaming store buffer holds a few (typically 4 or 8) cache lines augmented with presence bits for the parts of the line, such as for each word. The buffer sends a line to the directory when (1) the whole line is written (i.e., all presence bits for the line are set), (2) it is evicted by the replacement policy (we use round-robin), (3) a memory fence occurs, or (4) the core that issued the pushing store accesses the line with a normal load or store. Since streaming store buffers are similar to write combining buffers used for streaming stores [23], resources used for both buffers can be shared. Write combining buffers have been shown to be effective at reducing coherence traffic [13].

A line stored in the streaming store buffer is visible to the coherence domain only when the buffer sends it to the directory. Consequently, stores through the buffer are weakly ordered. Similarly, for unallocating loads and local pushing stores, streaming buffers are used to exploit spatial locality. The resulting deviation of memory ordering follows the already existing x86 streaming store semantics [22].

4. PROGRAMMING MODEL

The acceptance of new instructions is often determined by the programming system support. This section first discusses the portability of ULPS, and then discusses methods through which ULPS instructions can be generated.

4.1 Portability

Two aspects of portability are functional portability, or whether a piece of code will function correctly on another processor, and performance portability, or whether an optimized piece of code will be performant on another processor. Our instructions naturally provide functional portabil-

ity since they are identical to normal load/stores in how they modify architectural state. This greatly benefits programmers, compilers, or auto-tuners [11], who typically spend significant effort maintaining correctness.

Our instructions also provide performance portability by indirectly controlling the target cache level(s) through reuse distance hints. Specifying cache levels explicitly could result in inconsistent performance on processors with different cache hierarchies (i.e., number of levels and/or capacities). Since software provides reuse distance hints in registers (instead of as immediates), the values can also be input-dependent. Reuse distances can be profiled with low overhead using sampling techniques [14], and the profile data can predict reuse distances for different input data [12] and for caches with different capacities [14]. Although the basic definition of reuse distance considers memory accesses from a single thread, the concept can be extended to multi-thread and multi-programming environments. We can leverage existing work on analyzing reuse distance of parallel applications such as [50], which presents concurrent reuse distance for shared caches and private-stack reuse distance for private caches. Profiling techniques such as [39] can also predict effective shared cache capacities utilized by each thread or process, which allows us to compare (single-thread) reuse distances to the predicted effective capacities. When threads have a homogeneous memory access pattern, we can simply compare reuse distances to the shared cache capacity divided by the number of threads. This homogeneous access pattern is common in high performance computing domain, and we take this approach in our evaluation presented in §5.

One portability challenge is obtaining the target core IDs for remote pushing stores. We assume that the ISA and/or operating system provides a way to query for core ID—for example, in x86, this can be done with the `CPUID` instruction [22]. Software can keep a shared table with a mapping from application thread ID to core ID. The table could be created when threads are spawned and could be periodically updated in case threads migrate. If a thread does migrate, and the software uses a stale core ID, the program will still execute correctly, albeit slower and/or with lower energy efficiency until the ID is updated. Some run-time systems such as OpenMP may already maintain such a table. Alternatively, software can always use the `all_sharers` hint, as long as it ensures consumers peek at the addresses for incoming messages before the producers do their writes.

4.2 Code Generation

The compiler can generate our instructions with help from programmer annotation that extends the existing `nontemporal` pragma in the Intel compiler (ICC). For example, the code below uses the `nontemporal` pragma for a large array `A` to avoid cache pollution and wasteful allocations.

```
int *A = malloc(sizeof(int)*BIG);
...
#pragma nontemporal(A)
for (i = 0; i < BIG; i++)
  for (j = 0; j < LITTLE; j++) sum += A[i]*B[j];
```

This code is not performance-portable because the value `BIG` may be input-dependent or future architectures may have larger caches. We can annotate the pragma as `#pragma nontemporal(A(reuse_dist=BIG))`[2]. Once the reuse distance is available to the compiler, generating local ULPS instructions is no harder than generating streaming load/stores, which is already implemented in ICC.

For affine loops [15] commonly found in high-performance computing applications with regular memory accesses, even these simple programmer annotations are unnecessary. Production compilers including Portland Group Compilers™ and R-Stream® can analyze reuse distances for various loop transformations.

Similar programmer annotations can be used to generate remote pushing stores. For example, the code below pushes a portion of array `A` to the consumer core's cache.

```
#pragma nontemporal( \
  A+offset:LEN(reuse_dist=LEN, dest=consumer_tid))
for (i = 0; i < LEN; i++)
  A[offset + i] = ...
```

5. PERFORMANCE AND ENERGY EVALUATION

After describing our evaluation methodology, we verify the benefits of our approach using microbenchmarks, and then analyze the results on applications.

5.1 Evaluation Methodology

We implement ULPS in a modified Graphite simulator [29]. Table 5.1 shows the configurations of the simulated processor.

The processor utilizes a tile-based architecture; each tile contains a core, core-private L1, L2, and a directory slice, and the tiles are connected through a mesh interconnect. The core model is derived from the in-order core, out-of-order memory model described in [29]. Caches implement an ideal LRU replacement policy. For some experiments, we additionally model an eDRAM bank per tile. We modify our workloads to utilize ULPS in the same way a compiler would if the code were annotated as we discuss in §4.2.

[2]Strictly speaking, the reuse distance of each access to `A` is `BIG` + `LITTLE`, but using `BIG` as the reuse distance will give the same behavior as long as `BIG` is much bigger than `LITTLE`.

Table 2: Experiment Settings

Core	64 cores, dual-issue in-order x86 Core-private 32 KB 4-way L1-I$, 1 cycle Core-private 32 KB 8-way L1-D$, 1 cycle
Tile	1 core per tile Per-tile 512 KB 8-way L2$, 16 cycles, inclusive Directory slice, memory controller, and an eDRAM bank
Interconnect	2-D mesh, 1 physical network for coherence messages 2 cycles per hop (1 at router and 1 at link), 64-bit links
Directory	MOSI protocol, 20 cycles
eDRAM	512 MB, 64 banks, 16-way, 20 cycles
Memory	100 cycles
Compiler	Intel® Compiler (ICC)* 12.1.0e -O3 GCC 4.2.2 -O3 w/ libgomp

*Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

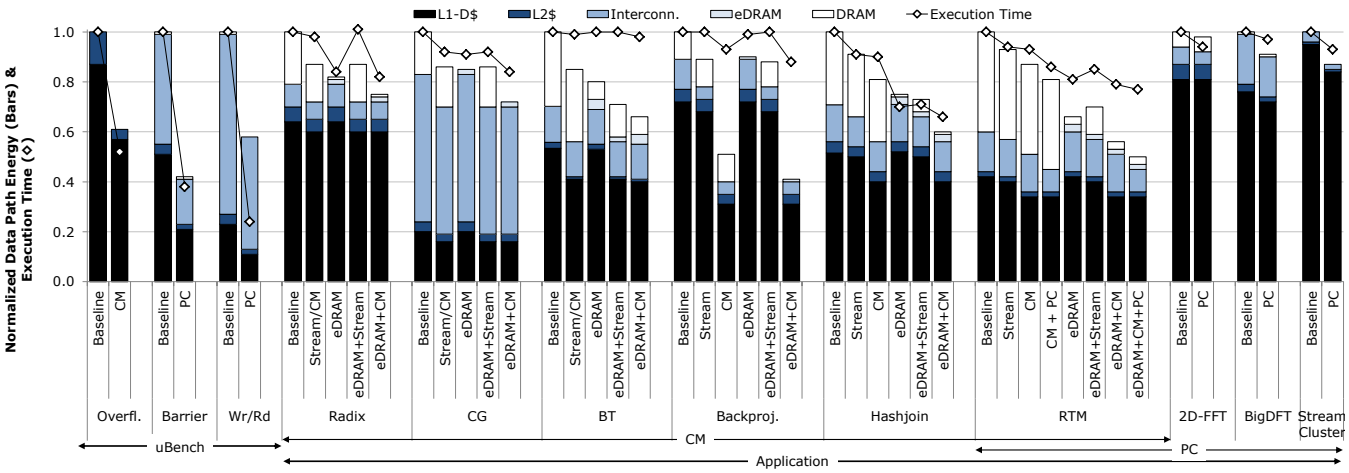


Figure 3: Energy Consumption Breakdown (Stacked Bars) and Normalized Execution Time (◇): `CM` = Cache-Level Management and `PC` = Producer-Consumer Communication.

To model interconnect energy, we use Orion [46] augmented with Graphite. To model cache and DRAM energy consumption, we augment Graphite with CACTI [30] version 6.5. Specifically, we pass activity counts from the simulator to CACTI to obtain dynamic energy for accesses to the directory and the cache tag and data arrays. We use the following parameters for CACTI: (1) "fast" mode for the L1-D cache that accesses tags and data in parallel, (2) "sequential" mode for the L2 cache and eDRAM that accesses tags and data sequentially, (3) high-performance devices for L1/L2, low standby power devices for eDRAM peripheral circuits, and commodity DRAM devices for eDRAM cells, all in 22 nm technology, and (4) eDRAM cells optimized for capacity with SDRAM-like interface as recommended in Thoziyoor et. al [44].

We do not model core energy, since it is highly dependent on implementation, and since our proposal will affect core energy mostly through reduced execution time. The memory subsystem consumes a significant fraction of a processor's power (>40% [43]), so reducing its energy consumption has a significant impact on processor energy.

5.2 Microbenchmark Results

Through microbenchmarks, we validate our implementation, and demonstrate the potential performance/energy impact of our proposal. The left side of Fig. 3 summarizes the measured improvements through ULPS and energy breakdowns in the microbenchmarks. Table 3 also lists statistics measured during the execution: cache miss rates, number of lines allocated in each cache, and the sum of the number of hops for all flits [10] sent. We provide data for multiple versions of each benchmark: `Baseline` and ULPS. The ULPS version(s) are labeled according to the usage of the instructions: `CM` for cache-level management, and `PC` for producer-consumer. The "ULPS Application" column in Table 1 shows optimizations we apply.

Overflow: `Overflow` repeatedly and sequentially accesses three arrays, although we skip elements so that we exploit no spatial locality. Each array is half as large as the L1-D, so most loads miss in the L1 (see Table 3). Of the three loads per loop iteration, two can be issued simultaneously in our dual-issue cores, while the third is serialized on the first two—each iteration combines the array elements with two arithmetic instructions, and the third load is embedded into the second arithmetic instruction (i.e., as a memory operand). As a result, each iteration takes ∼32 cycles (16 cycles per L2 access). The ULPS version uses an unallocating load for one of the three arrays to keep the array at the L2, and incurs L1-D misses only for that array. This reduces the L1-D miss rate to ∼33% and the cycles per iteration to 16, achieving a 2× speedup.

As for energy consumption, suppose the total number of loads is N. The baseline performs N L1-D allocations since almost all loads incur misses, while the ULPS version has very few L1-D allocations and $\frac{2}{3}N$ L1 reads. Since reads

Table 3: Workload Performance Statistics

Name	Config	Miss (%) L1-D	Miss (%) L2*	Line Alloc. L1-D	Line Alloc. L2	Flit Hops
`Overflow`	Baseline	100	0	46.2e5	∼0	∼0
	`CM`	34	0	0.3e5	∼0	∼0
`Barrier`	Baseline	2	2	37.3e3	38.8e3	4.9e6
	`PC`	0.1	0.1	1.3e3	3.0e3	2.0e6
`Wr/Rd`	Baseline	67	67	1.8e4	1.9e4	2.7e6
	`PC`	67	38	1.0e4	1.1e4	1.7e6
`Radix`	Baseline	2.2	2.2	5.2e7	5.1e7	8.5e8
	`Stream/CM`	2.2	2.1	3.5e5	1.2e5	6.0e8
`CG`	Baseline	38	27	2.2e9	1.6e9	18e9
	`Stream/CM`	38	25	1.8e9	1.1e9	16e9
`BT`	Baseline	6.8	5.8	9.6e7	7.6e7	1.3e9
	`Stream/CM`	6.2	5.7	6.1e7	4.9e7	1.2e9
`Backproj.`	Baseline	12	2.1	7.4e6	1.3e6	3.2e7
	Stream	11	1.8	6.2e6	5.3e4	1.3e7
	`CM`	6.2	1.8	8.7e2	5.3e4	1.3e7
`Hashjoin`	Baseline	19	6.9	4.7e6	1.7e6	2.3e7
	Stream	19	5.9	4.4e6	1.2e6	1.9e7
	`CM`	19	5.9	0.3e6	1.2e6	1.9e7
`RTM`	Baseline	19	7.6	12.4e7	4.7e7	1.6e8
	Stream	19	7.0	11.3e7	3.3e7	1.4e8
	`CM`	19	7.0	7.1e7	3.3e7	1.4e8
	`CM + PC`	20	5.9	6.9e7	2.7e7	1.2e8
`2D-FFT`	Baseline	2.2	1.5	6.9e5	4.4e5	6.7e7
	`PC`	2.2	1.1	5.5e5	3.1e5	5.2e7
`BigDFT`	Baseline	3.3	1.7	6.8e5	3.7e5	5.8e7
	`PC`	3.1	0.9	3.1e5	1.8e5	4.7e7
`Stream Cluster`	Baseline	8	0.4	20.0e5	2.1e5	4.5e7
	`PC`	7	0.2	18.0e5	0.1e5	3.1e7

∗ global miss rate = (L2 miss #)/(ld/st #) = (L1 local miss rate)×(L2 local miss rate)

and allocations consume roughly the same amount of energy according to our CACTI model, this yields a 1.5× energy reduction in the L1-D. At the L2, the baseline version reads N times, while the ULPS version reads $\frac{N}{3}$ times, resulting in a 3× L2 energy reduction.

Barrier: A combining tree barrier [51] has well-defined parent-children relations between threads, each of which acts as producer or consumer at each phase of the barrier. During the gather phase, children signal their parents by setting a `gather_flag`, on which their parent spin-waits. We use remote pushing stores to directly push the cache lines to the parent, avoiding ping-ponging.

During the scatter phase, parents signal their children through `scatter_flag`s. In invalidation-based coherence, the write by a parent invalidates its children, who then each needs to obtain a copy of the line. It is more efficient to signal the children in parallel, without invalidations. Since this is a single-producer multiple-consumer communication, we use pushing stores with an `all_sharers` hint to perform "multi-cast" pushing stores. This will update all children at once.

While the baseline has six messages in its signal propagation critical path, ULPS reduces it to two, so the theoretical maximum speedup is 3×. However, there are other sources of latency such as directory lookups and L2 accesses. Our analytical model, on the other hand, estimates the speedup from using ULPS as 2.6×[3], which matches our simulation.

Energy savings come from both the L1-D and interconnect. First, ULPS reduces the L1-D energy by 2.4×. This is from reduced spin waiting, where the reduction in L1-D accesses is similar to the execution time reduction. Second, for our coherence protocol, the baseline requires 44 messages per radix-4 group of threads, while the ULPS version reduces this to 10. However, the reduction in the number of flit hops is smaller (2.5× in Table 3), since some messages in the baseline are short control messages without data. The simulation shows a 2.4× energy reduction in the interconnect.

Wr/Rd: The `Wr/Rd` microbenchmark repeatedly executes two phases: (1) the producer creates a linked-list where each node occupies an entire cache line, and (2) the consumer traverses the list. The ULPS version utilizes pushing stores during phase (1) to proactively push the updated lines to the consumer's L2. It does this using an `all_sharers` hint. In phase (1), the ULPS version has a 2.4× faster average memory access time. This is due to each remote pushing store performing its write directly into the remote cache, rather than invalidating the remote cache, bringing the line into the local cache, and performing the write locally. In phase (2), the ULPS version has no L2 misses, resulting in a 6.5× faster average memory access time. Combining the two phases, the overall execution time improvement is 4.2×.

Energy-wise, while producers' fewer L1-D and L2 accesses and consumers' fewer L2 misses result in improvements, the majority of energy savings comes from the interconnect. Table 3 shows that the number of flit hops has been reduced by 1.6×, which matches the interconnect energy reduction.

[3]In this estimate, the head latency is computed using an average hop count of 5.3 from uniform random traffic in a 64-node mesh [10], and the contention delay is measured from simulation.

5.3 Application Results

The right side of Fig. 3 summarizes the application performance and energy improvements. ULPS instructions noticeably improve application performance. For our hardware configuration without eDRAM, the largest improvement is 1.16× (RTM), and the average speedup is 1.07× (excluding configurations with eDRAM). ULPS brings about even greater energy efficiency improvements—the average memory system energy reduction is 1.24×.

For those workloads where we apply cache-level management (`CM`), we also show the performance and energy obtained by applying conventional streaming load/stores (`Stream`). As discussed earlier, streaming load/stores only provide coarse-grained control: they can only specify that data should not be placed in *any* cache. We also evaluate configurations with eDRAM to illustrate the performance and energy improvements of ULPS compared to `Stream` in a deeper cache hierarchy (discussed in §5.4).

5.3.1 Cache-Level Management

Overall, when we use ULPS for cache-level management, we see performance and energy improvements from miss rate reductions at the targeted levels of cache, as well as energy improvements from fewer wasteful allocations for data that are not reused.

For `Radix`, `CG` and `BT`, we use unallocating loads and local pushing stores to avoid allocating large, streaming arrays in the cache hierarchy, to avoid cache pollution and wasteful cache line allocations. This allows us to capture more locality on smaller data structures at L1 and L2, resulting in slightly lower miss rates (see Table 3). Since `CG` is a memory bound application, the lower miss rates impact the performance more. For these applications, the energy used at L1-D and L2 is reduced, mostly from eliminating allocations for the arrays that do not fit in cache. A large fraction of energy is consumed in the interconnect in `CG` because the L2 miss rate is high (∼25%) and data is provided by remote L2 caches for most misses. Therefore, the reduced L2 miss rate also reduces the interconnect energy. For these three workloads, when eDRAM is not present, applying streaming loads and stores (i.e., `Stream`) provides the same improvement since our optimization is to avoid allocation throughout the entire hierarchy (hence, `Stream` and `CM` are shown together as `Stream/CM`).

`Backprojection` and `Hashjoin`, in contrast, exhibit more complex locality behavior. First, applying ULPS to the data structures that cannot fit in L2 reduces the L2 miss rate for both applications; in addition to saving L2 energy savings, this saves interconnect energy since some messages to the memory controller from L2 misses are eliminated. `Stream` sees these same benefits. However, we observe *additional* benefits for the applications when we apply unallocating loads to data structures that *do* fit in the L2, but not the L1, demonstrating the usefulness of finer-grained control of ULPS. Specifically, this optimization reduces wasteful L1 allocations, and for `Backprojection` it also eliminates some L1 pollution, greatly reducing the L1-D miss rate. For both benchmarks, bypassing the L1 for the working set that fits in L2 saves significant L1-D energy.

For `RTM`, let us first examine only the cache-level management versions. For these, `RTM` is similar to `Backprojection` and `Hashjoin`. We first reduce pollution in L2 by applying ULPS to the data structures that cannot fit in L2 (i.e.,

the same as Stream), but we further optimize the application by applying ULPS to data structures that fit in L2 but not L1. This reduces wasteful L1 allocations, providing an additional 13% L1-D energy reduction over Stream.

5.3.2 Producer-Consumer Communication

When we use remote pushing stores, we see performance gains when the communication is on the critical path, and we save energy throughout the cache hierarchy. We save interconnect energy from reduced on-die traffic (i.e., flit hops), cache energy from (a) eliminating invalidations and allocations in the producer's cache, and (b) reduced misses for the consumer. When accelerating synchronization, we also save L1-D energy from reduced spin waiting.

RTM is a stencil code; at each iteration, each thread pushes boundary values to neighbor threads' L2, reducing the L2 miss rate by 16% and the sum of flip hops by 14%. This in turn achieves 1.08× and 1.07× further improvements in performance and energy, respectively, compared to CM.

2D-FFT consists of row- and column-wise phases. At the end of the row-wise phase, we use pushing stores to place the data in the caches of threads that will read the data in the column-wise phase (we use core ID as a target hint since the consumers are not likely to be a current sharer). This reduces the sum of flip hops by 1.23×, yielding a 1.25× interconnect energy reduction. Using pushing stores also reduces the L2 miss rate by 1.25×, yielding a 1.06× speedup.

For BigDFT, we apply pushing stores in a similar fashion. This application performs a 3-D convolution via an x, y, and z phase. Between phases, we use pushing stores to proactively push the data to the consumer. The has a similar impact on the interconnect energy as 2D-FFT, but the performance improvement is smaller because accesses to the pushed data are not on the critical path.

StreamCluster, is barrier-intensive. Even with a moderate input size (simmedium), using the baseline tree barrier (see §2.2.2), the workload spends ∼30% of its execution time in the barrier. We drop-in replace the tree barrier with our pushing store enabled barrier. However, since the workload has inherent load imbalance (e.g., the main thread performs a reduction while the others are waiting), the barrier performance improvements are limited: on average, 1.31×. This improvement translates into a 1.07× application performance improvement. Energy reductions are similar to that explained in §5.2.

Other techniques such as prefetching and out-of-order execution might overlap communication latency with useful work. However, these techniques are often insufficient: access patterns may be too complex for prefetchers to capture, the consumer may not know when data is ready to prefetch, or latencies may be too large to hide. Hashjoin, Backprojection, and Barrier exhibit data-dependent random access patterns that are not easily predicted or overlapped, and RTM exhibits many simultaneous access streams that are not easily captured with prefetching [11].

5.4 Cache-Level Management with eDRAM

Figure 3 also reveals the power of finer-grained cache-level management as cache hierachies deepen. Here, we add an eDRAM cache as a large LLC, *but use the same application binaries tuned for the processor without the* LLC. Benefits from ULPS actually increase slightly, demonstrating the performance-portability of ULPS. In contrast, benefits from streaming load/stores either decrease or turn into liabilities. While the baseline binaries take advantage of the LLC, the binaries with streaming load/stores see smaller benefits since streaming load/stores bypass the LLC, even for data that can fit into the LLC. ULPS provides average speedups and energy savings of 1.06× and 1.34× respectively, over the baseline, and 1.11× and 1.33× over streaming load/stores.

For Radix, CG, and BT, the L2 cannot capture locality on large arrays, but the LLC can—for Radix and CG, eDRAM can fully contain the large arrays. However, eDRAM+Stream bypasses eDRAM for them, thereby consuming no less DRAM energy than Stream/CM. In fact, for Radix, when streaming load/stores are used for the large arrays, eDRAM captures no locality while adding latency from tag lookups; thus, eDRAM actually degrades the performance. In contrast, ULPS used in eDRAM+CM flexibly utilizes eDRAM to cache the large arrays, while avoiding pollution and wasteful allocations in L1 and L2. For BT, ULPS shows a similar flexibility, except that eDRAM+Stream consumes less DRAM energy than Stream/CM. This is because a few data structures are not managed by ULPS nor by streaming load/stores, and some of their locality can be captured by eDRAM.

Backprojection, Hashjoin, and RTM have more complex locality behavior, but see similar impact from the addition of eDRAM as the other applications: streaming load/stores and ULPS still improve L1 and L2 behavior, ULPS can additionally improve the LLC behavior, and streaming load/stores introduce inefficiency in the use of the LLC, actually increasing main memory activity versus the baseline.

5.5 Alternative Remote Pushing Store Implementations

In addition to optimizations described in §3.3, we can save invalidation messages when processing remote pushing stores. Suppose that core 2 was the only sharer in Fig. 2 (i.e., cores 2 and 3 are in fact the same). Then, the invalidation of the line in core 2 was unnecessary since it would be immediately followed by a reallocation with the updated data. Therefore, when the target is specified as a single core ID and that core already has exclusive ownership, no invalidation is necessary. Similarly, when an all_sharers hint is used, no invalidation is necessary since the lines in sharers' caches will be immediately updated, thus not affecting coherence.

However, when data are pushed to multiple cores, this optimization makes the memory consistency model depart from TSO. Nevertheless, we consider it a beneficial tradeoff because of the following reasons. First, for lazy remote pushing stores, we already deviated from TSO due to streaming store buffers explained before. Second, eager remote pushing stores are designed for synchronization, and will be mostly used while wrapped in constructs such as barriers, semaphores, and concurrent data structures. Designers of such synchronization constructs are typically well aware of the consequences of relaxed memory consistency and can use memory fences when needed.

We use Barrier to quantifies the benefits because it yields the biggest performance and energy improvements and because the optimization primarily affects eager remote pushing stores. While the performance and energy improvements from the optimized eager remote pushing stores with a relaxed consistency are 2.6× and 2.4× respectively, these reduce to 1.95× and 2.0× when TSO is preserved.

We consider yet another implementation that directly pushes the data to targets bypassing the directory. Although violating coherence, this gives the upper bound on improvements from saving coherence messages. We measure 3.6× and 3.4× energy and performance improvements (i.e., ours with relaxed consistency achieves 72% and 71% of this bound, while the TSO version achieves 54% and 59%). The gap comes from traversals to/from the directory, which can be closed by mechanisms that reduce the distances to home nodes.

6. RELATED WORK

6.1 Cache Data Placement Control

Streaming loads and stores provide a facility to avoid cache pollution and supply data directly to registers. In fact, streaming load/stores can be viewed as a special case of ULPS (identical to local ULPS with a very large reuse distance hint). Sections 5.3 and 5.4 demonstrated the benefit of having finer control over the levels data should reside. In particular, by adding a large LLC, we observe that the benefit from finer control becomes more important as cache hierarchies deepen.

Cache replacement policies optimized for streaming accesses [19, 24, 36, 49] are another way to control data placement. These still require wasteful allocations while ULPS avoids them at specified levels. These policies also require additional hardware to make predictions, such as additional state per line. For frequently accessed caches like L1, such hardware may impact latency and energy consumption. Some compiler algorithms [47] assist hardware to early evict lines with little locality. Instead of influencing replacement policies, these algorithms can also be used to generate ULPS instructions with the additional benefit of avoiding wasteful allocations.

Virtual local stores [8] try to reap the benefits of software-managed stores in conventional caches. However, only one contiguous address range per thread can be mapped to virtual local stores, leading to reduced flexibility. Cache-line locking [45] provides the ability to "pin-down" lines from being replaced with newly referenced lines. Column caching [7] provides a similar control but with the granularity of ways. Virtual local stores, cache-line locking, and column caching partition a cache and have the replacement policy treat a partition differently. As a result, the effectiveness of caches can be decreased from reducing the effective ways or sets.

As ULPS avoids destructive interference in caches from streaming and producer-consumer access patterns, cache partitioning schemes avoid destructive interference in shared caches from multiple workloads running simultaneously. These partitioning schemes such as utility-based cache partitioning [37] can help ULPS in multi-programming environments: specified reuse distances can be compared with the shared cache capacity allocated to each workload by the partitioner to decide whether a cache line should be allocated. NUCA optimizations such as adaptive selective replication [3] and elastic cooperative caching [20] improve the locality within a single cache level that is distributed over the chip. They can complement ULPS that optimizes for multiple cache levels.

6.2 Alternatives to Invalidation in Shared Memory

A number of previous schemes provide writers the ability to send the updated version of data to all current sharers rather than invalidating their copies, in a similar way to our remote pushing store with `all_sharers`. Update-based coherence protocols do this indiscriminately, which can waste significant bandwidth and energy. Dash's Write-Update mechanism [27], KSR1's Poststore instruction [38], Multicube's Notify instruction [17], and UltraComputer's UpdateAll instruction [4] are more selective in sending updates. All but Write-Update are instructions that the programmer or compiler inserts for specific writes.

Similarly, some previous schemes support writers explicitly sending data to specific processors, who may not already hold a copy of the cache line, in a similar way to our remote pushing store with specific core hints. Dash's Deliver instruction [27], the Write-and-Forward instruction [26], and remote stores [21] are all examples. With the exception of remote stores, these schemes force a producer to write data into its own cache as well as consumers'.

Unlike previous work, ULPS is flexible enough to allow both of these capabilities (updating existing copies of a line and sending data to specific cores). Additionally, ULPS gives software control over which level of cache to buffer messages in, to reduce pollution in consumers' caches.

6.3 Software-Managed Explicit On-chip Stores

Software-managed stores have been explored as an alternative to hardware-managed caches. Although software-managed stores increase the programmer's effort, prior works have shown the effectiveness for a set of workloads [18]. Scratch-pad memory [35] (SPM) is a small software-managed private store that is typically used in embedded systems and special purpose processors as a replacement for caches. SPMs as a replacement for caches are useful in embedded systems because only a few fixed applications are often fine-tuned with the underlying hardware. However, SPMs are typically not present in general purpose processors, because they pose hardware overheads and additional challenges to operating systems, and because not all applications benefit from their use.

In addition, Leverich et al. [28] show that software-managed buffers in streaming memory systems may not necessarily provide advantages. They observe that streaming at the programming model level instead is particularly beneficial to utilize prefetching and streaming stores in cache-based architectures, realizing benefits similar to streaming memory systems. ULPS provides software control finer than streaming stores, further improving the efficiency of cache-based architectures.

Nvidia's Fermi and Kepler architectures [33, 34] allow applications to determine the partitioning of their local store into cache and SPM. Gebhart [16] proposes a unified store architecture that can be repurposed for caches, SPMs, or register files. Similar to ULPS, his proposal allows flexibly using the same store through mechanisms optimized for different access patterns, but, unlike ULPS, accesses to caches, SPMs, and registers are not typically interchangeable without affecting correctness.

7. CONCLUSION

Allowing software control over on-die data placement can improve the efficiency of caches. We propose a mechanism to enable this, targeting better cache behavior for streaming and producer-consumer access patterns. Our *unallocating load* and *pushing store* instructions improve performance by

1.07× and reduce memory energy by 1.24× on average in a 64-core simulated system. Unlike other proposals for increasing software control over the memory hierarchy, ours is not disruptive, and instead naturally extends conventional cache architectures. Thus, our proposal provides benefits with small hardware changes, and is compatible with existing programming models.

Acknowledgements

The authors would like to thank Samantika Subramaniam and Rob F. Van der Wijngaart for discussion during the initial stage of our project. We thank Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Shubho Sengupta for some of the applications we evaluated, and Matthieu Ospici and Methieu Dubois at Bull for the BigDFT code. We also thank Pradeep Dubey for his guidance and anonymous reviewers who provided valuable comments.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood. Reverse Time Migration. *Geophysics*, 48(11), 1983.

[3] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *International Symposium on Micro-architecture (MICRO)*, 2006.

[4] Wayne Berke. A Cache Technique for Synchronization Variables in Highly Parallel, Shared Memory Systems. Technical Report Ultracomputer Note ♯151, Ultracomputer Research Laboratory, Courant Institute of Mathematical Sciences, December 1988.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[6] BigDFT Web-site. `http://inac.cea.fr/L_Sim/BigDFT`.

[7] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.

[8] Henry Cook, Krste Asanovic, and David A. Patterson. Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments. In *Tech. Rep. UCB/EECS-2009-131*, 2009.

[9] William J. Dally, François Labonté, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with Streams. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 35–42, 2003.

[10] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[11] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[12] Chen Ding and Yutao Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[13] Michel Dubois, Jin Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed Consistency and Its Effects On The Miss Rate of Parallel Programs. In *Proceedings Supercomputing '91*, 1991.

[14] David Eklov and Erik Hagersten. Statstack: Efficient Modeling of LRU caches. In *International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem Part I. One-dimensional Time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[16] Mark Alan Gebhart. *Energy-Efficient Mechanisms for Managing On-Chip Storage in Throughput Processors*. PhD thesis, The University of Texas at Austin, 2012.

[17] James R. Goodman, Mark K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[18] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[19] Mitchell Hayenga, Andrew Nere, and Mikko Lipasti. MadCache: A PC-aware Cache Insertion Policy. In *JILP Worshop on Computer Architecture Competitions: Cache Replacement Championship*, 2010.

[20] Enric Herrero, José González, and Ramon Canal. Elastic Cooperative Caching: An Autonomous Dynamically Adaptive Memory Hierarchy for Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[21] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. Remote Store Programming. In *International Conference on High-Performance Embedded Architectures and Compilers*, 2010.

[22] Intel 64 and IA-32 Architectures Software Developer Manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[23] Intel Corporation. Write Combining Memory Implementation Guidelines, 1998. `http://download.intel.com/design/PentiumII/applnots/24442201.pdf`.

[24] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*, 2010.

[25] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. In *International Conference on Very Large Data Bases (VLDB)*, 2009.

[26] David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12), 1996.

[27] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3), 1992.

[28] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 358–368, 2007.

[29] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[30] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *International Symposium on Micro-architecture (MICRO)*, pages 3–14, 2007.

[31] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[32] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–13, 2010.

[33] Nvidia. Fermi: Nvidia's next generation cuda compute architecture, 2009. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[34] Nvidia. Nvidia's next generation cuda compute architecture: Kepler gk110, 2012. `http://www.nvidia.com/content/PDF/kelper/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[35] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design and Test Conference*, pages 7–11, 1997.

[36] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *International Symposium on Computer Architecture (ISCA)*, pages 381–391, 2007.

[37] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *International Symposium on Micro-architecture (MICRO)*, 2006.

[38] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. In *SIGMETRICS*, 1993.

[39] Andreas Sandberg, David Black-Schaffer, and Erik Hagersten. Efficient Techniques for Predicting Cache Sharing and Throughput. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[40] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *International Conference on Management of Data (SIGMOD)*, 2010.

[41] Michael L. Scott and William N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. In *Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 44–52, 2001.

[42] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[43] Avinash Sodani. Race to Exascale: Opportunities and Challenges, MICRO'11 Keynote, 2011. `http://www.microarch.org/micro44/files/Micro%20Keynote%20Final%20-%20Avinash%20Sodani.pdf`.

[44] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *International Symposium on Computer Architecture (ISCA)*, 2008.

[45] Xavir Vera, Björn Lisper, and Jingling Xue. Data Cache Locking for Higher Program Predictability. In *SIGMETRICS*, 2003.

[46] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *International Symposium on Microarchitecture (MICRO)*, pages 294–305, 2002.

[47] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the Compiler to Improve Cache Replacement Decisions. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[48] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[49] W.A. Wong and J.-L. Baer. Modified LRU Policies for Improving Second-level Cache Behavior. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 49–60, 2000.

[50] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying Multicore Processor Scaling via Reuse Distance Analysis. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[51] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessors. *IEEE Transactions on Computers*, 36(4), 1987.